UNIVERSITY OF BORDEAUX

Mini-shell Report

Willian Ver Valem Paiva
git `https://github.com/WillianPaiva/projetSysteme2014`

November 19, 2014

1 TREE TRAVERSAL

at first to develop the shell I implemented a recursive function where it goes node by node to build up the command line in a way where it respect the pipes and redirections

for that i developed the following function:

Shell.c

```
int execute(Expression *e , int wait, int fdin,int fdout,int fderror, int lastflag, int
    futurewait)
{
    pid_t childPID;
    int fd;
    int pp[2];
    int mode;
    t_job* job;

    switch (e->type) {
        case SIMPLE:
            if(builtincommands(e) == 1){
                break;
            }else{
                childPID = fork();
                if(childPID >= 0) //fork was successful
                {
                    if(childPID == 0) //child process
                    {
                        if(fdin != 0){
                            dup2(fdin,0);
                            close(fdin);
```

```
                }
                if(fdout != 1){
                    dup2(fdout,1);
                    close(fdout);
                }
                if(fderror != 2){
                    dup2(fderror,2);
                    close(fderror);
                }
                for(int i = 3; i <= lastfd; i++){
                    close(i);
                }
                execvp(e->arguments[0], &e->arguments[0]);
                perror(e->arguments[0]);
                exit(1);
            }
            else//parent process
            {
                if(wait == 1){
                    mode = FOREGROUND;
                }else{
                    mode = BACKGROUND;
                }
                jobsList = insertJob(childPID,e->arguments[0], (int) mode,
                    futurewait);
                job = getJob(childPID, BY_PROCESS_ID);

                if(wait == 1){
                    for(int i = 3; i <= lastfd; i++){
                        close(i);
                    }
                    putJobForeground(job, FALSE);

                }else{
                    putJobBackground(job, FALSE);
                }
               // putchar('\n');
                break;
            }
        }
        else// fork failed
        {
            perror("fork");
        }
        break;
    }

case SEQUENCE:
    execute(e->gauche,1,fdin,fdout,fderror,0,0);
    execute(e->droite,1,fdin,fdout,fderror,0,0);
    break;
case SEQUENCE_ET:
    execute(e->gauche,0,fdin,fdout,fderror,0,1);
    execute(e->droite,1,fdin,fdout,fderror,0,0);
```

```
            break;
        case SEQUENCE_OU://needs better implementation
            execute(e->gauche,1,fdin,fdout,fderror,0,0);
            execute(e->droite,1,fdin,fdout,fderror,0,0);
            break;
        case BG:
            execute(e->gauche,0,fdin,fdout,fderror,0,0);
            break;
        case PIPE:
            if(pipe(pp) < 0){
                perror("pipe");
                exit(1);
            }
            ch_lastfd(pp[1]);
            execute(e->gauche,0,fdin,pp[1],fderror,0,1);
            if(lastflag == 1){
                execute(e->droite,1,pp[0],fdout,fderror,0,0);
            }else{
                execute(e->droite,0,pp[0],fdout,fderror,0,1);
            }
            break;
        case REDIRECTION_I:
            fd = open(e->arguments[0],O_RDONLY, 0666);
            ch_lastfd(fd);
            execute(e->gauche,1,fd,fdout,fderror,0,0);
            break;
        case REDIRECTION_O:
            fd = open(e->arguments[0],O_CREAT | O_RDWR, 0666);
            ch_lastfd(fd);
            execute(e->gauche,1,fdin,fd,fderror,0,0);
            break;
        case REDIRECTION_A:
            fd = open(e->arguments[0], O_TRUNC | O_CREAT | O_RDWR, 0666);
            ch_lastfd(fd);
            execute(e->gauche,1,fdin,fd,fderror,0,0);
            break;
        case REDIRECTION_E:
            fd = open(e->arguments[0], O_CREAT | O_RDWR, 0666);
            ch_lastfd(fd);
            execute(e->gauche,1,fdin,fdout,fd,0,0);
            break;
        case REDIRECTION_EO:
            fd = open(e->arguments[0], O_CREAT | O_RDWR, 0666);
            ch_lastfd(fd);
            execute(e->gauche,1,fdin,fd,fd,0,0);
            break;
        case VIDE:
            putchar('\n');
            break;
        default:
            break;

}
```

```
    return 0;
}
```

this function is the heart of the shell as it works his way up on the expression tree and create the appropriated forks and redirections. taking the recursive approach to deal with the expression tree made possible the interpretation of commands with n pipes and redirections.

for example:

```
ls | grep a | grep y | grep t > fich
cat fich

y.tab.c
y.tab.h
y.tab.o
```

2 BUILT-IN COMMANDS

the built-in commands were all implemented on the function :

Shell.c

```
int builtincommands(Expression *e)
{
    if(strcmp(e->arguments[0],"cd") ==0){
        if(e->arguments[1] != NULL){
            chdir(e->arguments[1]);
        }else{
            chdir(getenv("HOME"));
        }
        return 1;
    }
    if (strcmp(e->arguments[0],"jobs") ==0){
        printJobs();
        return 1;
    }
    if(strcmp(e->arguments[0],"exit") ==0){
        exit(EXIT_SUCCESS);
    }
    if(strcmp(e->arguments[0],"fg") ==0){
        if(e->arguments[1]== NULL){
            return 1;
        }
        int jobid = (int) atoi(e->arguments[1]);
        t_job* job = getJob(jobid, BY_JOB_ID);
        if(job == NULL){
            return 1;
        }
        if(job->status == SUSPENDED || job->status == WAITING_INPUT){
            putJobForeground(job,TRUE);
        }else{
```

```
            putJobForeground(job,FALSE);
        }
        return 1;
    }
    if(strcmp(e->arguments[0],"kill") ==0){
        if(e->arguments[1] == NULL){
            return 1;
        }else{
            killJob(atoi(e->arguments[1]));
            return 1;
        }
    }
    if(strcmp(e->arguments[0],"dirs") ==0){
        if(List_length(DirStack) >0){
            List_print(DirStack);
            return 1;
        }else{
            char p[1024];
            getcwd(p, sizeof(p));
            char *p2 = get_pwd(p);
            printf("%s\n",p2 );
            return 1;
        }
        return 1;
    }
    if(strcmp(e->arguments[0],"history") ==0){
        HIST_ENTRY **the_history_list =  history_list ();
        for(int i = 0;;i++){
            if(the_history_list[i] != NULL){
                printf("[%d]-------->   %s\n",i,the_history_list[i]->line);
            }else{
                break;
            }
        }
        return 1;
    }
    if(strcmp(e->arguments[0 ],"pushd") ==0){
        if(e->arguments[1] != NULL){
            char * checker = NULL;
            char path[1024] = "";
            char path2[1024] = "";
            checker = strstr(e->arguments[1], "/");
            if(checker == e->arguments[1])
            {
                strcat(path,e->arguments[1]);
            }else{
                getcwd(path, sizeof(path));
                strcat(path,"/");//you found the match
                strcat(path,e->arguments[1]);
            }
            struct stat s;
            if( stat(path,&s) == 0 )
            {
                if( s.st_mode & __S_IFDIR )
```

```
                {
                    getcwd(path2,sizeof(path2));
                    printf("%s\n",path );
                    chdir(path);
                    List_append(DirStack,path2);
                    return 1;
                }
                else if( s.st_mode & __S_IFREG )
                {
                    printf("%s   that is a file \n",path);//it's a file
                    return 1;
                }
            }
            else
            {
                perror("pushd");
            }

        }
        return 1;
    }
    if(strcmp(e->arguments[0],"popd") ==0){
        if(DirStack != NULL){
            printf("%s\n",List_get(DirStack,List_length(DirStack) -1));
            chdir(List_get(DirStack,List_length(DirStack) -1));
            List_pop(DirStack,List_length(DirStack) -1);
            return 1;
        }
        return 1;
    }
    if(strcmp(e->arguments[0],"printenv") ==0){
        char **en;
        for (en = env; *en != 0; en++)
        {
            char* thisEnv = *en;
            printf("%s\n", thisEnv);
        }
        return 1;
    }

    return 0;

}
```

The function execute calls builtincommands to verify if a command is type built-in if yes it execute the command else return 0 so execute can finish by executing it via exec.

for the commands *pushd popd* and *dirs* I implemnted a liked list to control the pile and use *chdir()* for changing directories:

Shell.c

```
Node *Node_create() {
    Node *node = malloc(sizeof(Node));
    assert(node != NULL);

    node->data = "";
    node->next = NULL;

    return node;
}


void Node_destroy(Node *node) {
    assert(node != NULL);
    free(node->data);
    free(node);
}


List *List_create() {
    List *list = malloc(sizeof(List));

    Node *node = Node_create();
    list->first = node;

    return list;
}


void List_destroy(List *list) {

    Node *node = list->first;
    Node *next;
    while (node != NULL) {
        next = node->next;
        free(node->data);
        free(node);
        node = next;
    }

    free(list);
}


void List_append(List *list, char *str) {
```

```
    Node *node = list->first;
    while (node->next != NULL) {
        node = node->next;
    }
    node->data = malloc(sizeof(char) * 1024);
    strcpy(node->data,str);
    node->next = Node_create();
}


void List_insert(List *list, int index, char *str) {
    assert(list != NULL);
    assert(str !=NULL);
    assert(0 <= index);
    assert(index <= List_length(list));

    if (index == 0) {
        Node *after = list->first;
        list->first = Node_create();
        list->first->data = str;
        list->first->next = after;
    } else if (index == List_length(list)) {
        List_append(list, str);
    } else {
        Node *before = list->first;
        Node *after = list->first->next;
        while (index > 1) {
            index--;
            before = before->next;
            after = after->next;
        }
        before->next = Node_create();
        before->next->data = str;
        before->next->next = after;
    }
}


char *List_get(List *list, int index) {
    if(list != NULL && 0 <=index && index < List_length(list)){
        Node *node = list->first;
        while (index > 0) {
            node = node->next;
            index--;
        }
        return node->data;
    }
    return "";
}


int List_find(List *list, char *str) {
    assert(list != NULL);
    assert(str != NULL);
```

```
    int index = 0;
    Node *node = list->first;
    while (node->next != NULL) {
        if (strlen(str) == strlen(node->data)) {
            int cmp = strcmp(str, node->data);
            if (cmp == 0) {
                return index;
            }
        }
        node = node->next;
        index++;
    }
    return -1;
}


void List_remove(List *list, int index) {
    assert(list != NULL);
    assert(0 <= index);
    assert(index < List_length(list));

    if (index == 0) {
        Node *node = list->first;
        list->first = list->first->next;
        Node_destroy(node);
    } else {
        Node *before = list->first;
        while (index > 1) {
            before = before->next;
            index--;
        }
        Node *node = before->next;
        before->next = before->next->next;
        Node_destroy(node);
    }
}


void List_pop(List *list, int index) {
    if(list != NULL && 0 <=index && index < List_length(list)){
        if (index == 0) {
            Node *node = list->first;
            list->first = list->first->next;
            char *data = node->data;
            Node_destroy(node);
        } else {
            Node *before = list->first;
            while (index > 1) {
                before = before->next;
                index--;
            }
            Node *node = before->next;
            before->next = before->next->next;
```

```
            char *data = node->data;
            Node_destroy(node);
        }
    }
}


int List_length(List *list) {
    int length = 0;
    if(list != NULL){
        Node *node = list->first;
        while (node->next != NULL) {
            length++;
            node = node->next;
        }

    }else{
        length = -1;
    }
    return length;
}


void List_print(List *list) {
    if(list != NULL){
        Node *node = list->first;
        while (node->next != NULL) {
            char *p2 = get_pwd(node->data);
            printf("%s\n", p2);
            node = node->next;
            if (node->next != NULL) {
            }
        }
    }else{
        printf("%s\n","empty stack" );
    }
}
```

- for the command *cd* i simple used the function *chdir()*.
- for *exit* just a *exit(EXIT_SUCCESS)*.
- for *kill* i send the signal *SIGKILL* to the child process.
- the command *history* was implemented by recovering the history already created by *GNU Readline* with i will talk about later.
- jobs and fg *see next section*

for the jobs control i created a linked list of jobs :

Shell.h

```
typedef struct job {
        int id;   //job id
        char *name; // command
        pid_t pid; //pid of the process
        int status; //status
        int futurewait; //a flag for waiting ore not
        struct job *next; //pointer to next job
} t_job;
```

all processes are inserted on the jobs list once a process is waited it is removed from the list that way the shell can control the foreground and background jobs via this 3 functions :

Shell.c

```
void putJobForeground(t_job* job, int continueJob)
{
    job->status = FOREGROUND;
    if (continueJob) {
        if (kill(job->pid, SIGCONT) < 0)
            perror("kill (SIGCONT)");
    }
    waitJob(job);
}

void putJobBackground(t_job* job, int continueJob)
{
    if(job->status == FOREGROUND){
        job->status = BACKGROUND;
    }
    if (job == NULL){
        return;
    }
    if (continueJob && job->status != WAITING_INPUT){
        job->status = WAITING_INPUT;
    }
    if (continueJob){
        if (kill(job->pid, SIGCONT) < 0){
            perror("kill (SIGCONT)");
        }
    }
}


void waitJob(t_job* job)
{
    int terminationStatus;
    actualJob = job->pid;
```

```
    while (waitpid(job->pid, &terminationStatus, WNOHANG) == 0) {
        if (job->status == SUSPENDED)
            return;
    }
    jobsList = delJob(job); //remove job after wait
    actualJob = 0;

}
```

the second part of the jobs control is implemented by a *signal handler* where it defines the status of a job and defines what action take. this *signal handler* catches just the *SIGCHLD*.

Shell.c

```
void sigchild_handler(int sig)
{
    pid_t pid;
    int terminationStatus;
    pid = waitpid(-1, &terminationStatus, WUNTRACED | WNOHANG);
    if (pid > 0) {
        t_job* job = getJob(pid, BY_PROCESS_ID);
        if (job == NULL){
            return;
        }
        if (WIFEXITED(terminationStatus)) {
            if (job->status == BACKGROUND) {
                //printf("\n[%d]+  Done\t    %s\n", job->id, job->name);
                jobsList = delJob(job);
            }
        } else if (WIFSIGNALED(terminationStatus)) {
            printf("\n[%d]+  KILLED\t   %s\n", job->id, job->name);
            jobsList = delJob(job);
        } else if (WIFSTOPPED(terminationStatus)) {
            if (job->status == BACKGROUND) {
                changeJobStatus(pid, WAITING_INPUT);
                printf("\n[%d]+   suspended [wants input]\t   %s\n",
                        numActiveJobs, job->name);
            } else {
                changeJobStatus(pid, SUSPENDED);
                printf("\n[%d]+   stopped\t   %s\n", numActiveJobs, job->name);
            }
            return;
        } else {
            if (job->status == BACKGROUND) {
                jobsList = delJob(job);
            }
        }
    }

}
```

once I started to work on the signals I was faced with a problem. once the *shell* received a signal the *Lex* crashed , so i went into the *Lex* code and found out that the problem resided on the function *YY_INPUT* after some research i found a solution by overwriting *YY_INPUT* with another function using *GNU Readline*. with the implementation of this I got auto-completion, recursive search, history and other features bash like. the implementation was made by those modifications:

Analise.l

```
#undef  YY_INPUT
#define YY_INPUT(buf,result,max_size) \
                rl_input((char *)buf, &result, max_size)
```

Shell.h

```
static void rl_input (buf, result, max)
        char *buf;
        int  *result;
        int   max;
{

  if (rl_len == 0)
    {
                getcwd(cwd, sizeof(cwd));
        char *cwd2 = get_pwd(cwd);
                if(getuid()==0){
                        user = "#";
                }else{
                        user = "$";
                }
        char hostname[1024];
        hostname[1023] = '\0';
        gethostname(hostname, 1023);


                char ps1[1024]="␣" ; //definition of a prompt for the shell
                strcat(ps1,"\x1b[33m");
                strcat(ps1,"[");
                strcat(ps1,"\x1b[36m");
                strcat(ps1,getUserName());
                strcat(ps1,"\x1b[33m");
                strcat(ps1,"@");
                strcat(ps1,"\x1b[34m");
                strcat(ps1,hostname);
                strcat(ps1,"\x1b[33m");
                strcat(ps1,"]");
                strcat(ps1,"\x1b[32m");
                strcat(ps1,"␣␣");
                strcat(ps1,cwd2);
                strcat(ps1,"␣");
                strcat(ps1,"\x1b[34m");
                strcat(ps1,user);
                strcat(ps1,"\x1b[0m");
```

```
      if (rl_start)
        free(rl_start);
      rl_start = readline(ps1);
      if (rl_start == NULL) {
        /* end of file */
        *result = 0;
        rl_len = 0;
        return;
      }
      rl_line = rl_start;
      rl_len = strlen(rl_line)+1;
      if (rl_len != 1)
        add_history(rl_line);
      rl_line[rl_len-1] = '\n';
      fflush(stdout);
    }

  if (rl_len <= max)
    {
      strncpy(buf, rl_line, rl_len);
      *result = rl_len;
      rl_len = 0;
    }
  else
    {
      strncpy(buf, rl_line, max);
      *result = max;
      rl_line += max;
      rl_len -= max;
    }
}
```